KNN regression experiments

In class we learned about how KNN regression works, and tips for using KNN. For example, we learned that data should be scaled when using KNN, and that extra, useless predictors should not be used with KNN. Are these tips really correct?

In this notebook we run a bunch of tests to see how KNN is affect by the choice of k, scaling of the predictors, presence of useless predictors, and other things.

One experiment we do not run, and which would be interesting, is to see how KNN performance changes as a function of the size of the training set.

INSTRUCTIONS

Enter code wherever you see # YOUR CODE HERE in code cells, or YOU TEXT HERE in markup cells.

In [253…
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor
import matplotlib.pyplot as plt
```

In [254…
```
# set default figure size
plt.rcParams['figure.figsize'] = [8.0, 6.0]
```

In [255…
```
# code in this cell from:
# https://stackoverflow.com/questions/27934885/how-to-hide-code-from-cells-in-ip
from IPython.display import HTML

HTML('''<script>
code_show=true;
function code_toggle() {
 if (code_show){
 $('div.input').hide();
 } else {
 $('div.input').show();
 }
 code_show = !code_show
}
$( document ).ready(code_toggle);
</script>
<form action="javascript:code_toggle()"><input type="submit" value="Click here t
```

Out[255…
```
Click here to display/hide the code.
```

Read the data and take a first look at it

The housing dataset is good for testing KNN because it has many numeric features. See

Aurélien Géron's book titled 'Hands-On Machine learning with Scikit-Learn and TensorFlow' for information on the dataset.

In [256…

```
df = pd.read_csv("https://raw.githubusercontent.com/grbruns/cst383/master/housin
```

In [257…

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object 
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Note that numeric features have different ranges. For example, the mean value of *'total_rooms'* is over 2,500, while the mean value of *'median_income'* is about 4. *'median_house_value'* has a much greater mean value, over $200,000, but we will be using it as the target variable.

In [258…

```
from IPython.display import Image
from IPython.core.display import HTML
Image(url= "data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAAQABAAD/2wCEAAoHCBIVFRgVF
#Flickr source for "Houses going down" : https://www.flickr.com/photos/59937401@
```

Out[258…

In [259…

```
df.describe()
```

Out[259…

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	po
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	po
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682

Missing Data

Notice that 207 houses are missing their *total_bedroom* info:

In [260…

```
print(df.isnull().sum())
df[df['total_bedrooms'].isnull()]
```

```
longitude               0
latitude                0
housing_median_age      0
total_rooms             0
total_bedrooms        207
population              0
households              0
median_income           0
median_house_value      0
ocean_proximity         0
dtype: int64
```

Out[260…

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	house
290	-122.16	37.77	47.0	1256.0	NaN	570.0	
341	-122.17	37.75	38.0	992.0	NaN	732.0	
538	-122.28	37.78	29.0	5154.0	NaN	3741.0	
563	-122.24	37.75	45.0	891.0	NaN	384.0	
696	-122.10	37.69	41.0	746.0	NaN	387.0	
...	...	...	...	...	...	...	
20267	-119.19	34.20	18.0	3620.0	NaN	3171.0	
20268	-119.18	34.19	19.0	2393.0	NaN	1938.0	
20372	-118.88	34.17	15.0	4260.0	NaN	1701.0	
20460	-118.75	34.29	17.0	5512.0	NaN	2734.0	
20484	-118.72	34.28	17.0	3051.0	NaN	1705.0	

207 rows × 10 columns

Let's drop these instances for now

In [261…

```
df = df.dropna()
```

Prepare data for machine learning

We will use KNN regression to predict the price of a house from its features, such as size, age and location.

We use a subset of the data set for our training and test data. Note that we keep an unscaled version of the data for one of the experiments we will run.

In [262…

```
# for repeatability
np.random.seed(42)
```

In [263…

```
# select the predictor variables and target variables to be used with regression
predictors = ['longitude','latitude','housing_median_age','total_rooms', 'total_
#dropping categortical features, such as ocean_proximity, including spatial ones
target = 'median_house_value'
X = df[predictors].values
y = df[target].values
```

In [264…

```
# KNN can be slow, so get a random sample of the full data set
indexes = np.random.choice(y.size, size=10000)
X_mini = X[indexes]
y_mini = y[indexes]
```

In [265…

```
# Split the data into training and test sets, and scale
scaler = StandardScaler()

# unscaled version (note that scaling is only used on predictor variables)
X_train_raw, X_test_raw, y_train, y_test = train_test_split(X_mini, y_mini, test

# scaled version
X_train = scaler.fit_transform(X_train_raw)
X_test = scaler.transform(X_test_raw)
```

In [266…

```
# sanity check
print(X_train.shape)
print(X_train[:3])
```

```
(7000, 8)
[[ 1.22783551 -1.3492796   0.34639424 -0.16627017  0.11697691 -0.15874461
   0.18687025 -0.74984935]
 [ 0.62095726 -0.82169566  0.58720859 -0.11584049 -0.22077651 -0.0770853
  -0.14171346  1.12877289]
 [-1.16983102  0.7563873  -0.45632025 -0.32112946  0.02736886 -0.37395092
  -0.04890738 -0.10303138]]
```

Baseline performance

For regression problems, our baseline is the "blind" prediction that is just the average value of the target variable. The blind prediction must be calculated using the training data. Calculate and print the test set root mean squared error (test RMSE) using this blind prediction. I have provided a function you can use for RMSE.

In [267…

```
def rmse(predicted, actual):
    return np.sqrt(((predicted - actual)**2).mean())
```

In [268…

```
knn = KNeighborsRegressor()
knn.fit(X_train, y_train)
predictions = knn.predict(X_test).mean()
result = rmse(predictions,y_test)
print("test, rmse baseline: " + str(result))
```

```
test, rmse baseline: 112900.81843155583
```

Performance with default hyperparameters

Using the training set, train a KNN regression model using the ScikitLearn KNeighborsRegressor, and report on the test RMSE. The test RMSE is the RMSE computed using the test data set.

When using the KNN algorithm, use algorithm='brute' to get the basic KNN algorithm.

In [269…

```
knn = KNeighborsRegressor(algorithm='brute')
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
result = rmse(predictions,y_test)
print("test RMSE, default hyperparameters " + str(result))
```

```
test RMSE, default hyperparameters 62448.852862157735
```

Impact of K

In class we discussed the relationship of the hyperparameter k to overfitting.

_I provided code to test KNN on k=1, k=3, k=5, ..., k=29. For each value of k, compute the training RMSE and test RMSE. The training RMSE is the RMSE computed using the training data. Use the 'brute' algorithm, and Euclidean distance, which is the default. You need to add the get_train_test*rmse() function.*

In [270…

```
def get_train_test_rmse(regr, X_train, X_test, y_train, y_test):
    regr.fit(X_train, y_train)
    predictions = regr.predict(X_test)
    predictions_test = regr.predict(X_train)
    return rmse(predictions_test, y_train), rmse(predictions, y_test)
```

In [271…

```
n = 30
test_rmse = []
train_rmse = []
ks = np.arange(1, n+1, 2)
for k in ks:
    print(k, ' ', end='')
    regr = KNeighborsRegressor(n_neighbors=k, algorithm='brute')
    rmse_tr, rmse_te = get_train_test_rmse(regr, X_train, X_test, y_train, y_tes
    train_rmse.append(rmse_tr)
    test_rmse.append(rmse_te)
print('done')
```

```
1  3  5  7  9  11  13  15  17  19  21  23  25  27  29  done
```

In [272…

```
# sanity check
print('Test RMSE when k = 3: {:0.1f}'.format(test_rmse[2]))
```

```
Test RMSE when k = 3: 62448.9
```

Using the training and test RMSE values you got for each value of k, find the k associated with the lowest test RMSE value. Print this k value and the associated lowest test RMSE value. In other words, if you found that k=11 gave the lowest test RMSE, then print the value 11 and the test RMSE value obtained when k=11.

In [273…

```
def get_best(ks, rmse):
    best_c = rmse.index(min(rmse))
    best_rmse = rmse[best_c]
    return best_c*2+1, best_rmse

best_k, best_rmse = get_best(ks, test_rmse)
print('best k = {}, best test RMSE: {:0.1f}'.format(best_k, best_rmse))
```

```
best k = 7, best test RMSE: 62421.5
```

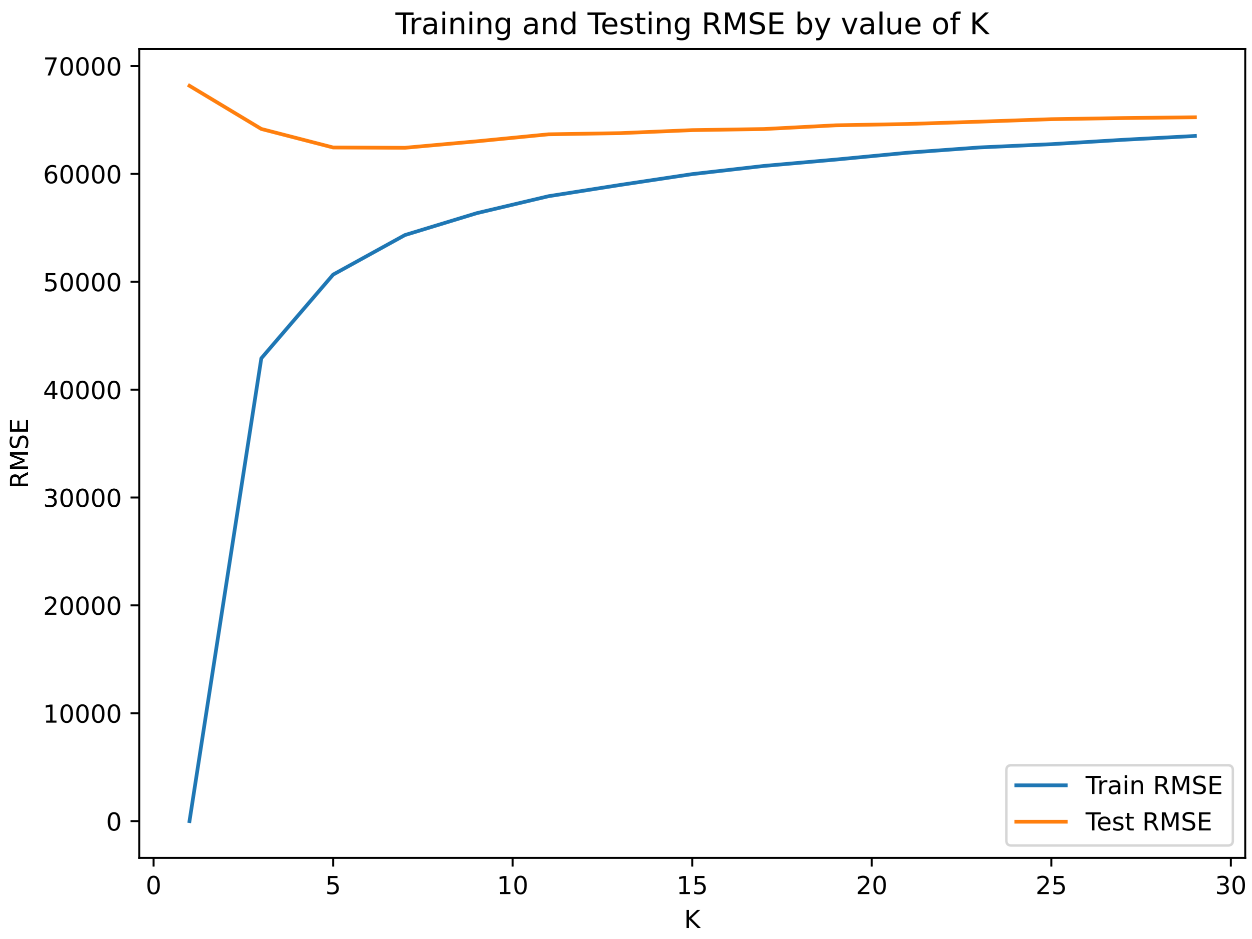
Plot the test and training RMSE as a function of k, for all the k values you tried.

In [274…

```
plt.plot(ks, train_rmse)
plt.plot(ks, test_rmse)
plt.title("Training and Testing RMSE by value of K")
plt.xlabel("K")
plt.ylabel("RMSE")
plt.legend(['Train RMSE', "Test RMSE"])
```

Out[274…

```
<matplotlib.legend.Legend at 0x7ff960dbcc10>
```



Comments

In the markup cell below, write about what you learned from your plot. I would expect two or three sentences, but what's most important is that you write something thoughtful.

I mainly had a good experience with trying out different values and experimenting with different inputs. I do notice when the k values go higher the RMSE values will as well.

Impact of noise predictors

In class we heard that the KNN performance goes down if useless "noisy predictors" are present. These are predictor that don't help in making predictions. In this section, run KNN regression by adding one noise predictor to the data, then 2 noise predictors, then three, and then four. For each, compute the training and test RMSE. In every case, use k=10 as the k value and use the default Euclidean distance as the distance function.

_The add_noise_predictor() method makes it easy to add a predictor variable of random values to X_train or X*test.*

In [275…

```
def add_noise_predictor(X):
    """ add a column of random values to 2D array X """
    noise = np.random.normal(size=(X.shape[0], 1))
    return np.hstack((X, noise))
```

_Hint: In each iteration of your loop, add a noisy predictor to both X_train and X_test. You don't need to worry about rescaling the data, as the new noisy predictor is already scaled. Don't modify X_train and X*test however, as you will be using them again.*

In [276…

```
noisy_x_train = X_train.copy()
noisy_x_test = X_test.copy()
noisy_test_rmse = []
for x in range(5):
    print(f'{x}', end=' ')
    #train
    regr = KNeighborsRegressor(n_neighbors=10, algorithm='brute')
    rmse_tr, rmse_te = get_train_test_rmse(regr, noisy_x_train, noisy_x_test, y_
    noisy_test_rmse.append(rmse_te)
    # add noise
    noisy_x_train = add_noise_predictor(n_x_train)
    noisy_x_test = add_noise_predictor(n_x_test)
print("done")
```

```
0 1 2 3 4 done
```

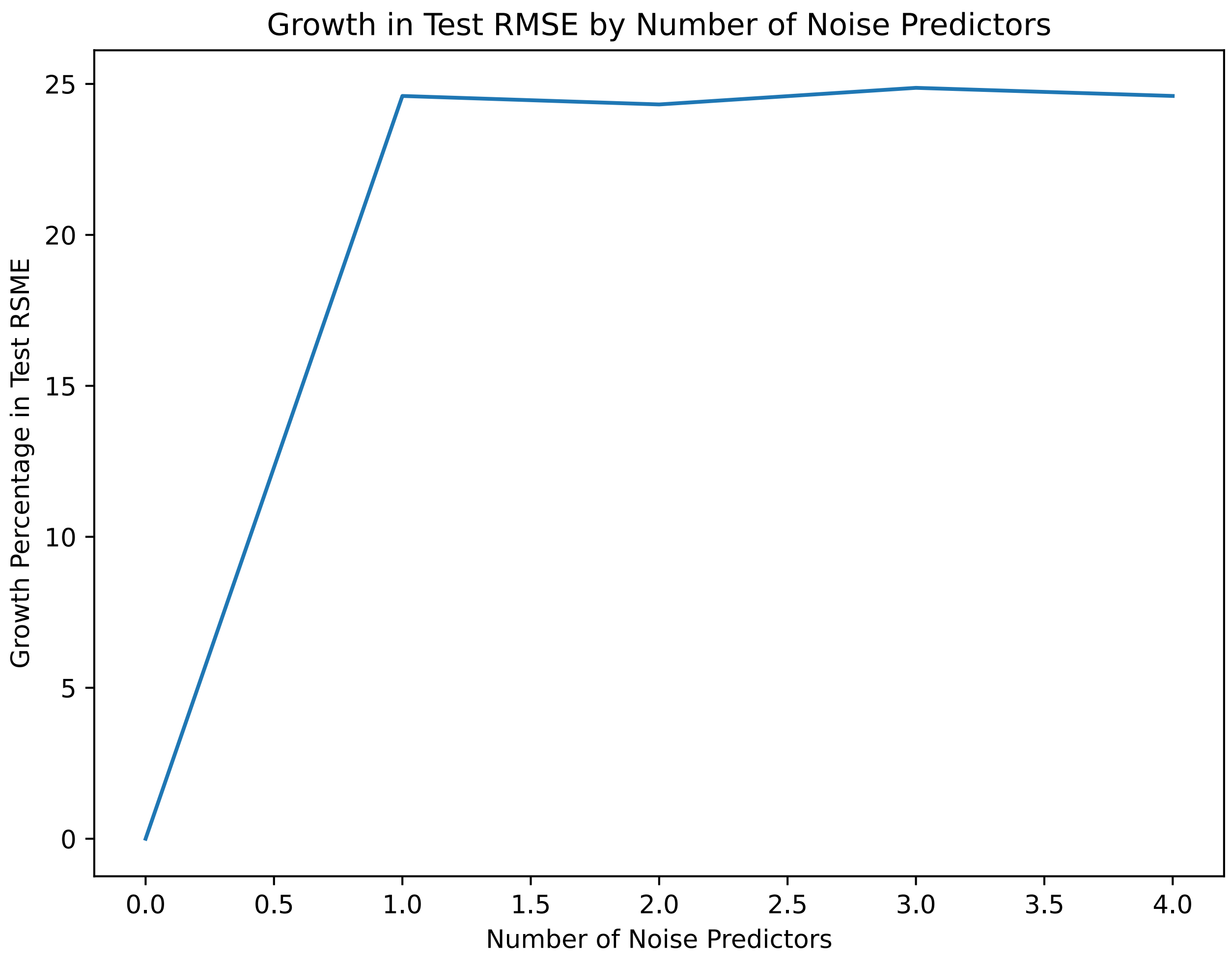
Plot the percent increase in test RMSE as a function of the number of noise predictors. The x axis will range from 0 to 4. The y axis will show a percent increase in test RMSE.

_To compute percent increase in RMSE for n noise predictors, compute 100 * (rmse - base_rmse)/base_rmse, where base*rmse is the test RMSE with no noise predictors, and rmse is the test RMSE when n noise predictors have been added.*

In [277…

```
base_rmse = noisy_test_rmse[0]
noise_cols = np.arange(5)
rmse_growth = []
for x in noisy_test_rmse:
 rmse_growth.append(100 * (x-base_rmse)/base_rmse)

plt.plot(noise_cols,rmse_growth)
plt.title("Growth in Test RMSE by Number of Noise Predictors")
plt.xlabel('Number of Noise Predictors')
plt.ylabel('Growth Percentage in Test RSME')
plt.show()
```



Comments

Look at the results you obtained and add some thoughtful commentary.

Impact of scaling

In class we learned that we should scaled the training data before using KNN. How important is scaling with KNN? Repeat the experiments you ran before (like in the impact of distance metric section), but this time use unscaled data.

Run KNN as before but use the unscaled version of the data. You will vary k as before. Use algorithm='brute' and Euclidean distance.

In [278…

```
n = 30
test_rmse_raw = []
train_rmse_raw = []
ks = np.arange(1, n+1, 2)
for k in ks:
 print(k, ' ', end='')
 regr = KNeighborsRegressor(n_neighbors=k, algorithm='brute', metric='euclidean'
 rmse_tr, rmse_te = get_train_test_rmse(regr, X_train_raw, X_test_raw, y_train,
 train_rmse_raw.append(rmse_tr)
 test_rmse_raw.append(rmse_te)
print('done')
```

```
1  3  5  7  9  11  13  15  17  19  21  23  25  27  29  done
```

Print the best k and the test RMSE associated with the best k.

In [279…

```
best_k, best_rmse = get_best(ks, test_rmse_raw)
print('best k = {}, best test RMSE: {:0.1f}'.format(best_k, best_rmse))
```

```
best k = 9, best test RMSE: 94057.4
```

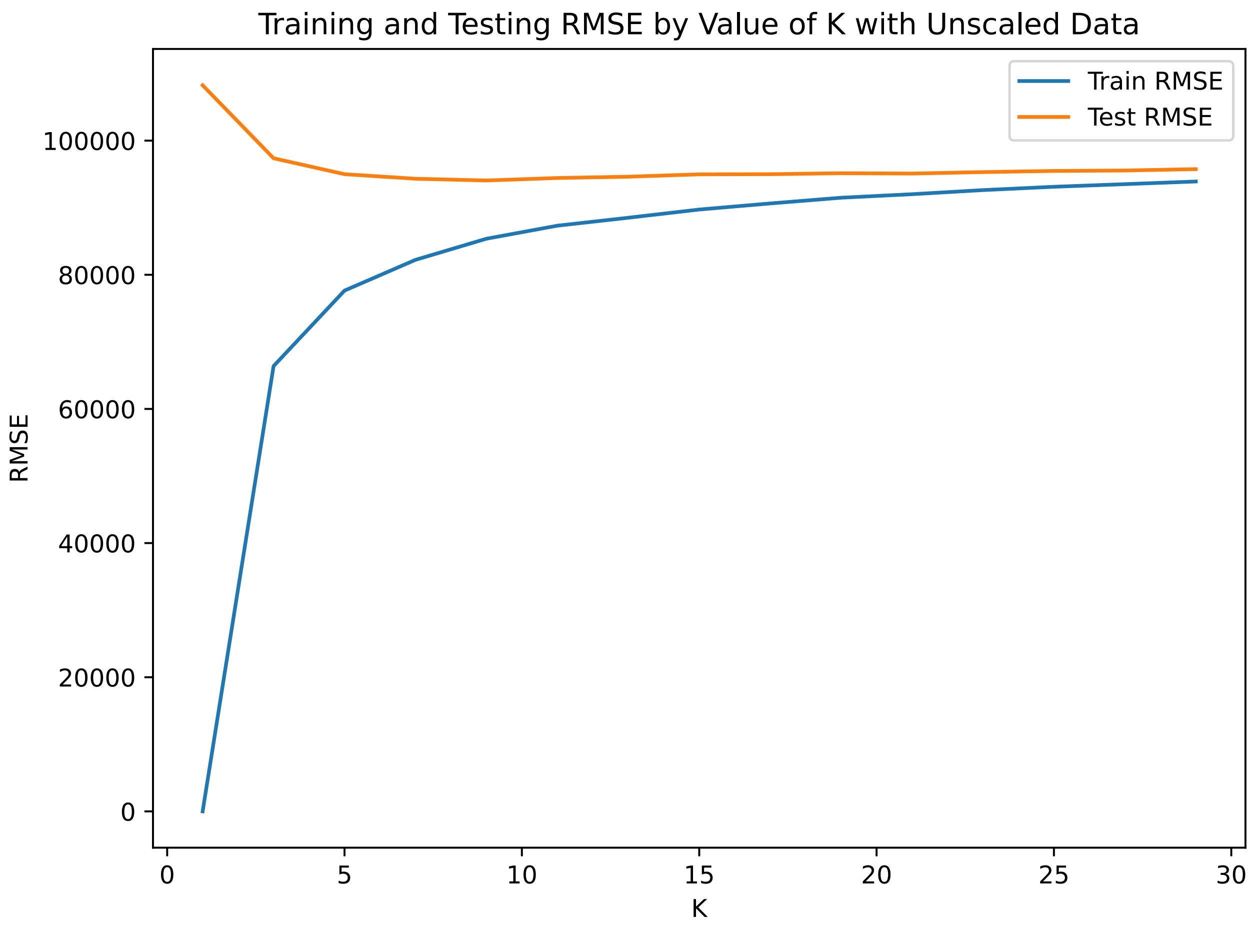
Plot training and test RMSE as a function of k. Your plot title should note the use of unscaled data.

In [280…

```
plt.plot(ks, train_rmse_raw)
plt.plot(ks, test_rmse_raw)
plt.title("Training and Testing RMSE by Value of K with Unscaled Data")
plt.xlabel("K")
plt.ylabel("RMSE")
plt.legend(['Train RMSE', "Test RMSE"])
```

Out[280…

```
<matplotlib.legend.Legend at 0x7ff953b17970>
```



Comments

Reflect on what happened and provide some short commentary, as in previous sections.

The main thing that I learned from this was that the graphs were mostly indentical without much changes.

Impact of algorithm

We didn't discuss in class that there are variants of the KNN algorithm. The main purpose of the variants is to be faster and to reduce that amount of training data that needs to be stored.

_Run experiments where you test each of the three KNN algorithms supported by Scikit-Learn: ball_tree, kd*tree, and brute. In each case, use k=10 and use Euclidean distance.*

In [281…

```
k = 10
algorithms = ['brute', 'ball_tree', 'kd_tree']
regr_cont = pd.Series(0.0, index=algorithms)
for algo in algorithms:
 print(algo, end=' ')
 regr = KNeighborsRegressor(n_neighbors=k, algorithm=algo)
 regr.fit(X_train, y_train)
 predictions = regr.predict(X_test)
 rsme_val = rmse(predictions, y_test)
 regr_cont.update(pd.Series(rsme_val, index=[algo]))
print('done')
```

```
brute ball_tree kd_tree done
```

Print the name of the best algorith, and the test RMSE achieved with the best algorithm.

In [282…

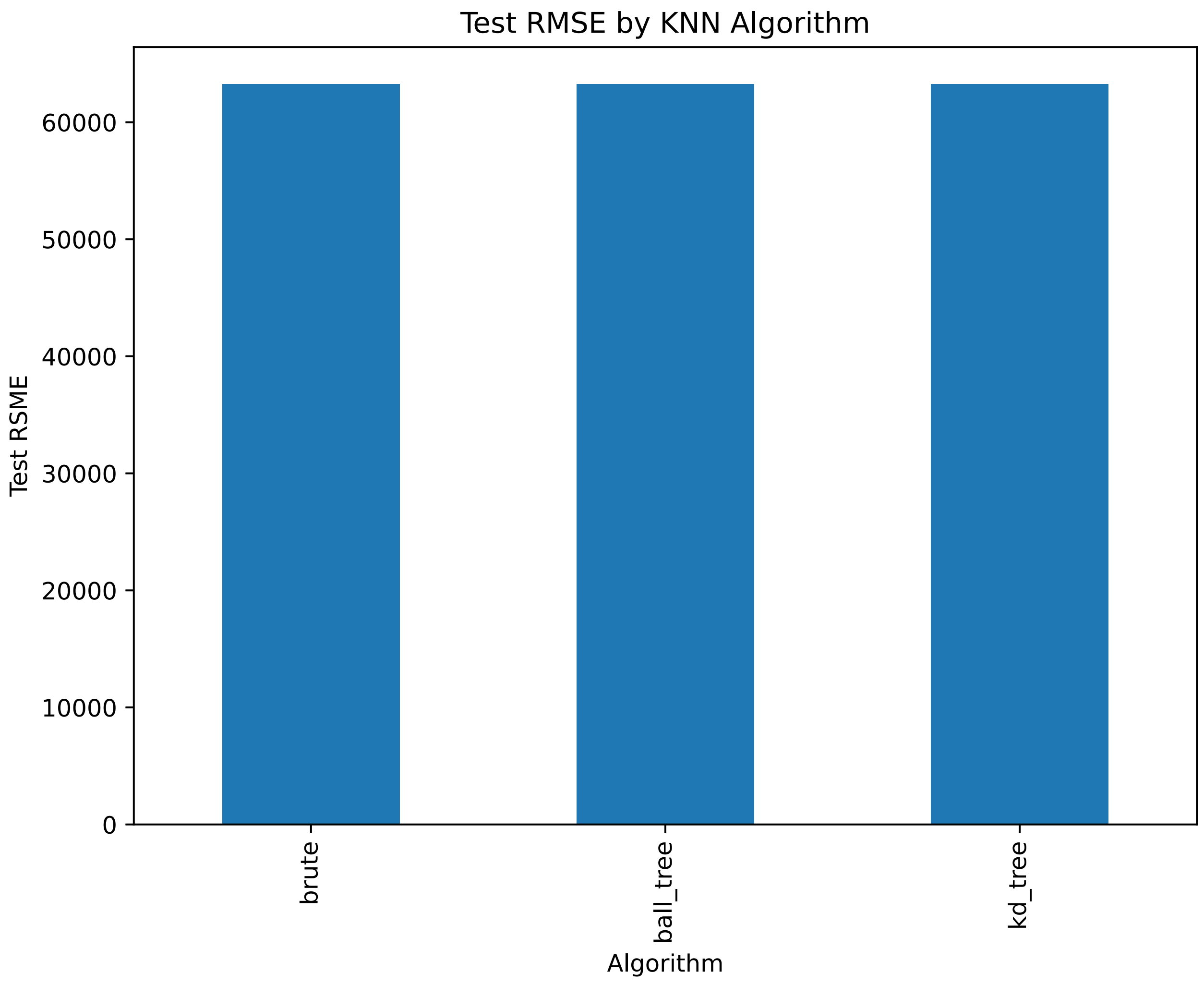
```
print(f'The best algorithm: \'{regr_cont.idxmin()}\' | RSME: {regr_cont.min():.3
```

```
The best algorithm: 'brute' | RSME: 63254.845
```

Plot the test RMSE for each of the three algorithms as a bar plot.

In [283…

```
regr_cont.plot.bar()
plt.title('Test RMSE by KNN Algorithm')
plt.xlabel('Algorithm')
plt.ylabel('Test RSME')
plt.show()
```



Comments

As usual, reflect on the results and add comments.

It appears the best was the brute algorithm but I'm a little confused on my plotting, I'm not sure if I'm intereperting this data correctly.

Impact of weighting

It was briefly mentioned in lecture that there is a variant of KNN in which training points are given more weight when they are closer to the point for which a prediction is to be made. The 'weight' parameter of KNeighborsRegressor() has two possible values: 'uniform' and 'distance'. Uniform is the basic algorithm.

Run an experiment similar to the previous one. Compute the test RMSE for uniform and distance weighting. Using k = 10, the brute algorithm, and Euclidean distance.

In [284…

```
k = 10
weights = ['uniform', 'distance']
regr_cont = pd.Series(0.0, index=weights)
for w in weights:
 print(w, end=' ')
 regr = KNeighborsRegressor(n_neighbors=k, weights=w)
 regr.fit(X_train, y_train)
 predictions = regr.predict(X_test)
 rsme_val = rmse(predictions, y_test)
 regr_cont.update(pd.Series(rsme_val, index=[w]))
print('done')
```

```
uniform distance done
```

Print the weighting the gave the lowest test RMSE, and the test RMSE it achieved.

In [285…

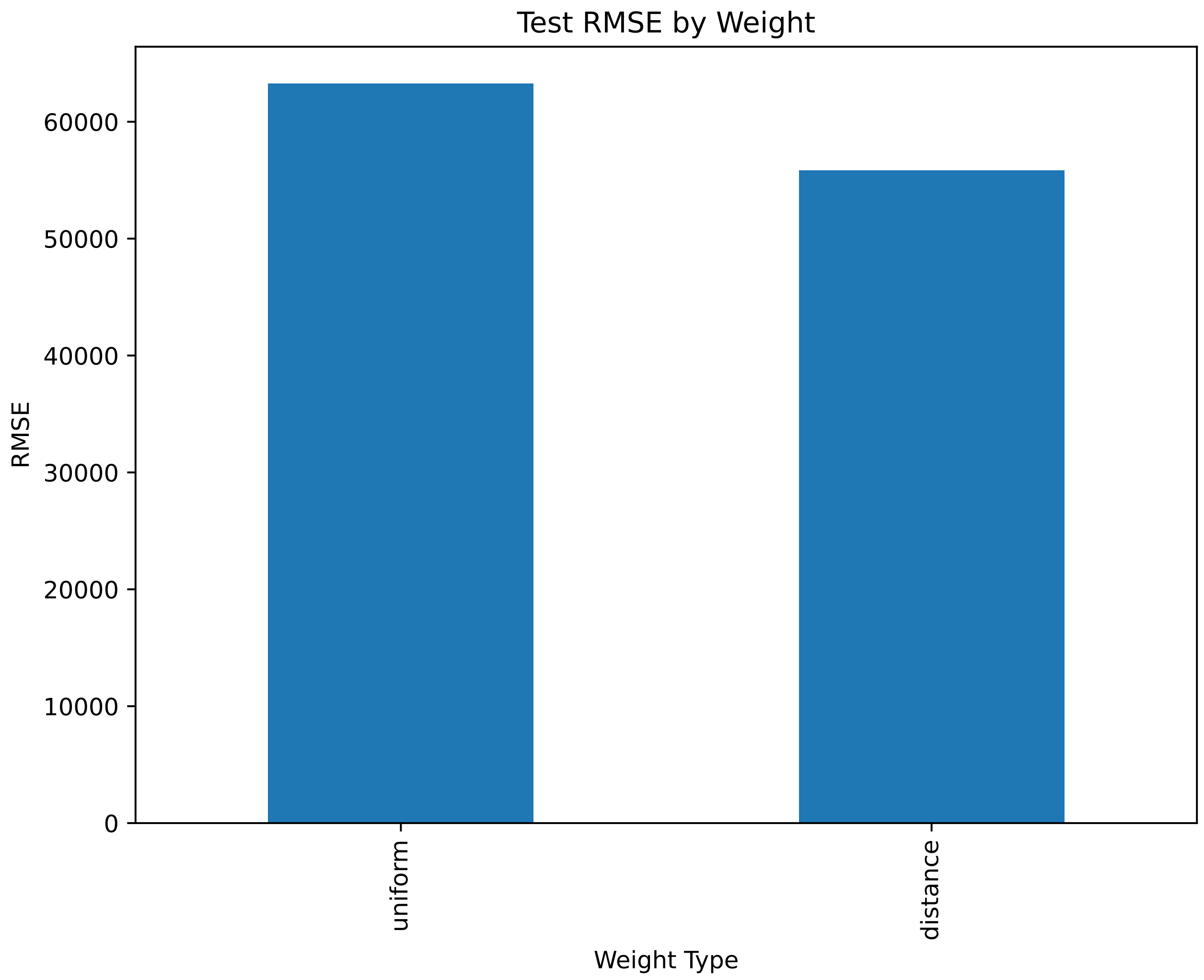
```
print(f'The best weight: \'{regr_cont.idxmin()}\' | RMSE: {regr_cont.min():.3f}'
```

```
The best weight: 'distance' | RMSE: 55817.065
```

Create a bar plot showing the test RMSE for the uniform and distance weighting options.

In [286…

```
regr_cont.plot.bar()
plt.title('Test RMSE by Weight')
plt.xlabel('Weight Type')
plt.ylabel("RMSE")
plt.show()
```



Comments

As usual, reflect and comment.

Results show as the best weight with an RMSE: 55817.065 changing weight makes in smaller RMSE. I did feel that the results were very simliar and plotting the data helped me visualize it better.

Conclusions

Please provide at least a few sentences of commentary on the main things you've learned from the experiments you've run.

A lot of my experience from this lab was trial and error to get values to be shown correctly. A few instances I was expecting different results but I tried to learn from what the data shown/plotted for me. For a few questions, I wasn't sure if I was getting the correct results and I had to go back to modules to see if I was doing things correctly. The results were challenging to read correct/incorrect data.